Math.

UIUCDCS-R-80-1016

UILU-ENG 80 1716

PATHOS: A Path Pascal Operating System

by

Martin S. McKendry
Roy H. Campbell
Robert B. Kolstad

June 1980

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-80-1016

PATHOS: A Path Pascal Operating System

by

Martin S. McKendry
Roy H. Campbell
Robert B. Kolstad

June 1980

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

Supported in part by NASA Project NSG 1471 and NSF MCS 77-09128.

1 Introduction.

Several mechanisms for implementation of the sequencing, exclusion, and encapsulation problems peculiar to operating systems have been proposed [Hewitt & Atkinson, 79] [Hoare, 74]. One of the difficulties in evaluating these mechanisms is the measurement of such intangibles as learnability, ease of use, and utility for use in production environments. Nonetheless, experience with programming languages indicates that these factors have as much bearing on the likely success of a language as academic evaluations; a language which encourages elegant solutions to textbook problems may not suit the complexities found in actual use. This paper discusses the principles of Path Pascal [Campbell & Kolstad, 79] and examines their application to the construction of an example operating system.

Pathos (Path Operating System) is a portable interactive system which supports the production and execution of Path Pascal programs at the application level while particularly facilitating network and teletype interfaces at the device level [McKendry, 80]. The system characterizes the set of applications for which Path Pascal was designed. The usual problems of process synchronization, mutual exclusion, message passing, and device handling arise and are solved by objects written in Path Pascal. These situations are sufficiently complex and interesting that an indication of the suitability of the language for general operating systems construction can be obtained.

In accordance with current design philosophies [Dijkstra, 68] [Liskov, 72], Pathos is implemented as a hierarchy of processes. So that the specific issues addressed by Path Pascal may be emphasized, Pathos is somewhat simpler than a general operating system and is restricted to three levels (user, super-

visor and kernel). The three levels are distinct in their environments and responsibilities. Application and interactive programs are at the user level. Supporting the user level is the system (supervisor) level: a set of processes coded in Path Pascal which provide services that are required at the user level but not available to the system level (such as logical I/O and memory management). The kernel process is at the lowest level of the system. It provides functions required by the system level, including scheduling of system level processes, procedure entry and exit code, and run time management of Path Expressions.

The system level of Pathos, which resembles the canonical message-oriented system described in [Lauer & Needham, 78], is a network of processes which communicate by exchanging messages. This paper focuses on the message passing subsystem of the system level, taking as main points of interest Path Pascal's influence on design of system structure and its benefits in the areas it was designed to assist: synchronization, encapsulation, and program organization. The implementation of the design was straightforward and free of many of the pitfalls encountered when traditional synchronization mechanisms are used.

The remainder of this paper presents Path Pascal and the principles of its use, then provides an overview of Pathos' structure and an examination of the program structures used to implement that structure. The paper concludes with a discussion of the results of the Pathos implementation.

2 Path Pascal.

Pathos is implemented in Path Pascal, a high level programming language which includes objects for encapsulation, processes which execute independently, path expressions for synchronization, and provisions for coding interrupt

processes. Path Pascal allows code for synchronization and coordination of asynchronous systems to be written entirely in a high level language. It is currently implemented on several computers: the CDC Cyber family, the PDP-11 family, the Z80 microprocessor, the PRIME 650, and the IBM Series 1.

2.1 Objects.

Pascal permits data structures to be created from arrays or records of other data structures. Path Pascal has an additional structuring mechanism called an object. The object is similar to a Pascal record except that no standard access operations are provided to manipulate its internal representation. User defined operations are written as entry routines in the form of procedures, functions or processes. The object includes a Path Expression to specify the synchronization between executions of these operations. Objects can form types or be used directly in variable declarations. An implementation of a stack is shown below:

```
CONST stacksize = 235;

TYPE stack = OBJECT
    PATH stacksize: (push; pop), 1:(push, pop) END;

    VAR stk: ARRAY [1..stacksize] OF element;
        pointer: 0..stacksize;

    ENTRY PROCEDURE push (VAR item: element);
        BEGIN pointer := pointer + 1;
        stk [pointer] := item;
        END;

    ENTRY FUNCTION pop: element;
        BEGIN pop := stk [pointer];
        pointer := pointer - 1;
        END;

        INIT; BEGIN pointer := 0; END;

    END (* stack *)
```

The object declaration consists of the header, Path Expression, local variable declarations, entry procedures and optional initialization block. The internal data structure contained in variables of type 'stack' ('pointer' and 'stk') cannot be accessed outside of the body of the object stack. Thus all manipulation of variables of type 'stack' must be performed by invocations of the operations 'push' and 'pop'. The values of stack are determined by these definitions and the order in which execution of the operations may occur. In this case, the Path Expression defines all the legal sequences of operations that can be performed on the stack data structure without contravening its stack-like behavior. The Path Expression specifies that there are 'stacksize' resources to be shared between executions of the procedures 'push' and 'pop'. The 'push' operation acquires a resource and the 'pop' operation releases one. The second part of the Path Expression (after the first comma) declares that a single resource (the stack pointer) is to be shared between 'push' and 'pop'. Path Expressions are described in more detail below.

Variables of type stack may be created by the declarations shown below:

```
VAR stk1: stack;
    stka: ARRAY [1..n] OF RECORD
                      f1: stack;
                      f2: stack;
                      END;
```

The declaration of an object type creates a variable which is an instance of that type and which can be referenced in program statements by using the variable's identifier. Each instance contains its own synchronization information stored independently from other instances of the same type. Object types may be used to construct other types and data structures and may be nested within other objects, records or arrays. Statements which request operations on objects use a Simula-like dot notation to indicate the variable of type object

and the operation to be invoked. Two invocation requests are shown below:

```
stk1.push (token);
with stka[1] do f1.push (f2.pop);
```

Frequently the data structure of a type requires initialization. Path Pascal allows an object to be initialized using a special init block. This block is executed when an object is instantiated. In the stack example above, the init block initialized the stack pointer.

Objects may include constant and type definitions. The constants and types may be used in the formation of the internal data structure of the object or to provide abstract data types which are managed by the object. An abstract data type which is to be managed by an object is declared within that object as an entry type. Entry types are exported to the scope containing the object and may be used in type and variable declarations. The internal data structure of the entry type is completely inaccessible outside the object.

2.2 Path Expressions.

Normally the order of invocation of procedures is unknown until the invocation occurs since processes can execute asynchronously. Path Expressions are used in Path Pascal for the specification of synchronization constraints on procedures contained within objects. Path Expressions allow three distinct kinds of constraints to be specified: sequencing (denoted by ';'), resource restriction (denoted by 'n:()'), and resource derestriction (denoted by '[ ]'). Resource derestriction is not used in Pathos. It is discussed in [Campbell & Kolstad, 79]. Each of these can be combined with other forms to provide complex

synchronization constraints and several constraints can be contained in a single Path Expression. These forms are described with examples below.

A path with no synchronization information consists of a comma separated list of routine names. The path below

PATH name1, name2, name3 END

imposes no restriction on the order of invocation of the routines nor any restriction on the number of concurrent executions of 'name1', 'name2' and 'name3'.

The sequencing mechanism allows the imposition of an order on procedure invocations. The desired order is specified by a semi-colon separated list. In the example below:

PATH first; second; third END

one execution of routine 'first' must complete before each invocation of 'second' may begin, and one execution of 'second' must complete before each invocation of 'third' can begin. Of course, the current execution of a 'third' or 'second' in no way inhibits the initiation of 'first' -- several routines may be executing concurrently.

Often it is necessary to limit the number of concurrent executions of a routine. The resource restriction specification allows concurrent execution of routines to proceed until the restriction limit is reached. Restrictions are denoted by surrounding the expression to be restricted by parentheses and preceding it with the integer restriction limit and a colon. The restriction below:

PATH 2:(ttyhandler) END

allows only two invocations of 'ttyhandler' to proceed concurrently. Any routine invoking 'ttyhandler' must then wait until less than two invocations are active before it can begin execution. A critical section, in which only a single resource is to be shared is easily specified. In the example below:

```
PATH 1:(routine1, routine2, routine3) END
```

only one of the three routines can be active at a time.

Each of the forms above (without PATH END) can be considered to be a subexpression of a Path. Each of these subexpressions may be combined (with optional use of parentheses for clarity) in the formats above to form complex paths. The sequencing operator has higher precedence than the separators. Multiple constraints specified for a given routine must all be satisfied before the routine can begin execution.

2.3 Concurrent Processes.

Path Pascal also provides a facility for asynchronous execution of several different processes, sequences of actions whose locus of control is contained within a local domain. These processes may be thought of as executing concurrently and no assumptions may be made about their order of execution. The processes may operate on shared, encapsulated data by calling the associated entry routines of the data's object. Processes may also operate on local and global data. The process's own data (and that of its surrounding scopes) may be manipulated freely since Pascal scope rules still apply. Process declarations differ from procedure declarations only in that the declarator process is substituted for the declarator procedure.

3 Principles of Programming in Path Pascal.

Wirth [Wirth, 77] states that "it is natural to refine program and data specifications in parallel." Path Pascal extends this principle to include specification of not only programs and data but also of synchronization. Path Pascal allows simple stepwise refinement of program flow, data structures, and synchronization.

For program flow, Path Pascal contains the standard high level control mechanisms used in structured programming. Considerable literature has addressed this subject and it will not be discussed here. Data refinement is enhanced through the use of the Path Pascal object. This construct allows the isolation, encapsulation, and abstraction of shared data structures. It is possible to specify operations on data with complete confidence that only those operations specified can manipulate the data. Programs can execute objects' operations with no regard to the data's physical instantiation, with the guarantee that the operations provide correct manipulation of the data.

Access to shared data (which normally must be guarded against inadvertent concurrent or incorrectly sequenced manipulations) is protected through Path Expressions. The sequencing operator (';') blocks processes attempting to access data before their turn while the restrictor operator (':') constrains the maximum number of concurrent executions within its scope. The Path Expression provides a central, static description of the synchronization requirements of the object's routines. No variables or conditions are necessary to completely specify normally required synchronizations. All synchronization constructs have single entry and exit points. These 'proper' synchronization schemes [Zelkowitz, et al., 79] enhance provability and intellectual manageability in a

similar fashion to the elimination of the GOTO statement from sequential programs.

With these mechanisms it is possible to conceive of a concurrent system as a set of cooperating concurrent processes with communication through synchronized, shared data (contained in objects). The clear separation of data, manipulations of the data, synchronization, and program flow enhances modularity and modifiability. These criteria are precisely those specified by Parnas as desirable in the decomposition of systems into modules [Parnas, 72].

4 Operating System Design.

The design philosophy adopted for Pathos led to its level structure being further constrained to a tree (figure 1). Each process implements services for its descendants and has available to it those services implemented by its ancestors.

The kernel process is at the root of the tree and runs on the bare machine, assuming only those facilities provided by the hardware. The current Path Pascal kernel for the PDP-11 is coded in assembly language and is 1000 lines (3500 bytes) long. The kernel supports the system level by providing primitive services such as Path synchronization, procedure entry and exit code, and simple scheduling. The system level in turn supports the user level by performing most of the duties commonly associated with operating systems such as file system support, memory management, and processor allocation. Finally, the user processes are at the bottom of the process tree and utilize all the services implemented at supporting levels.

Within the system level only one process is responsible for providing

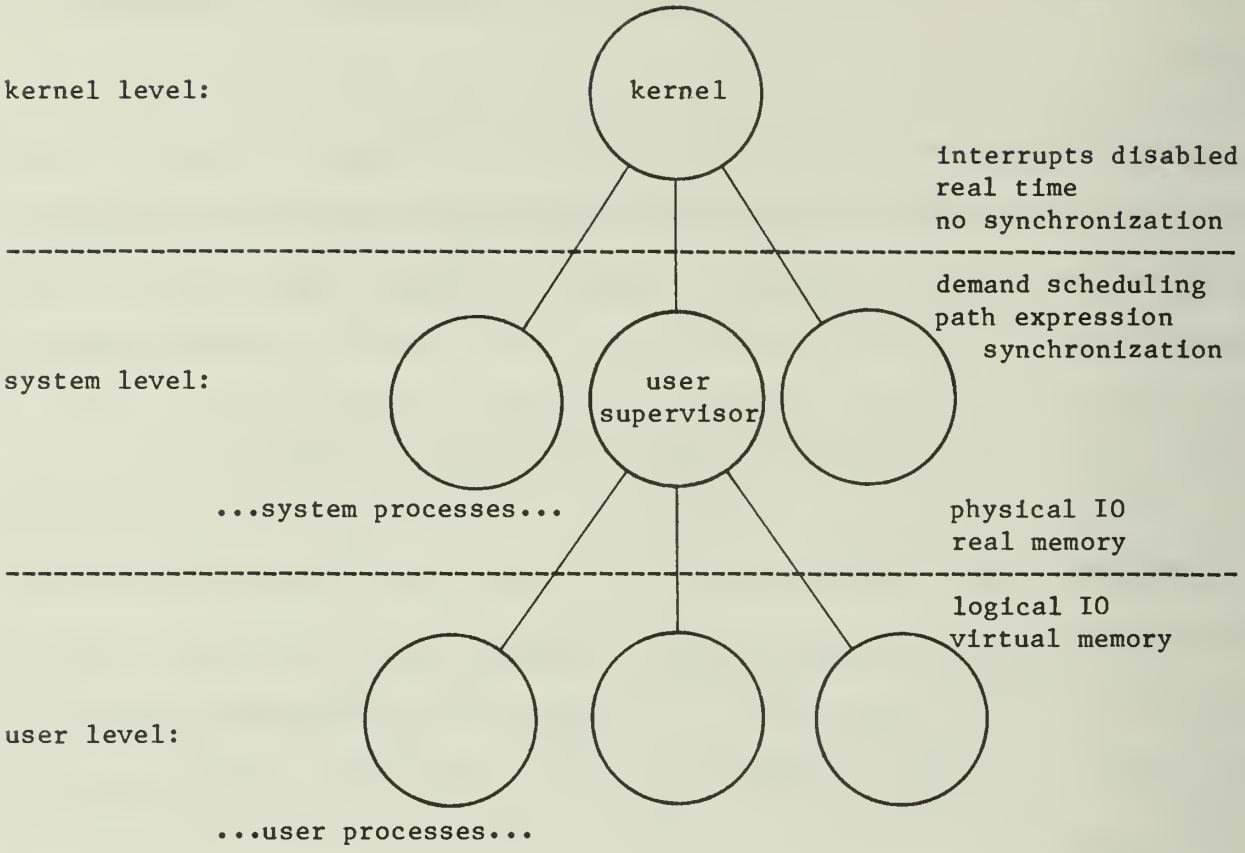

Figure 1: Process Configuration.

each of the major services. Thus one process manages the file system, another process manages user level requests (the 'user supervisor') and so on. User requests are mapped into requests to other processes in the form of messages. These messages are passed between processes by the message passing system which is managed by the 'communication controller'. The communication controller is also the 'main line' of the operating system source program and consequently instantiates all other system level processes.

Communication between processes is implemented by a 'port' assigned to each process at its instantiation. The port object supervises all message traffic to or from its assigned process. It provides operations such as 'put' and 'get' which pass messages to and receive messages from other processes. A

port is logically divided into 'channels', each channel having a distinct destination port (figure 2).

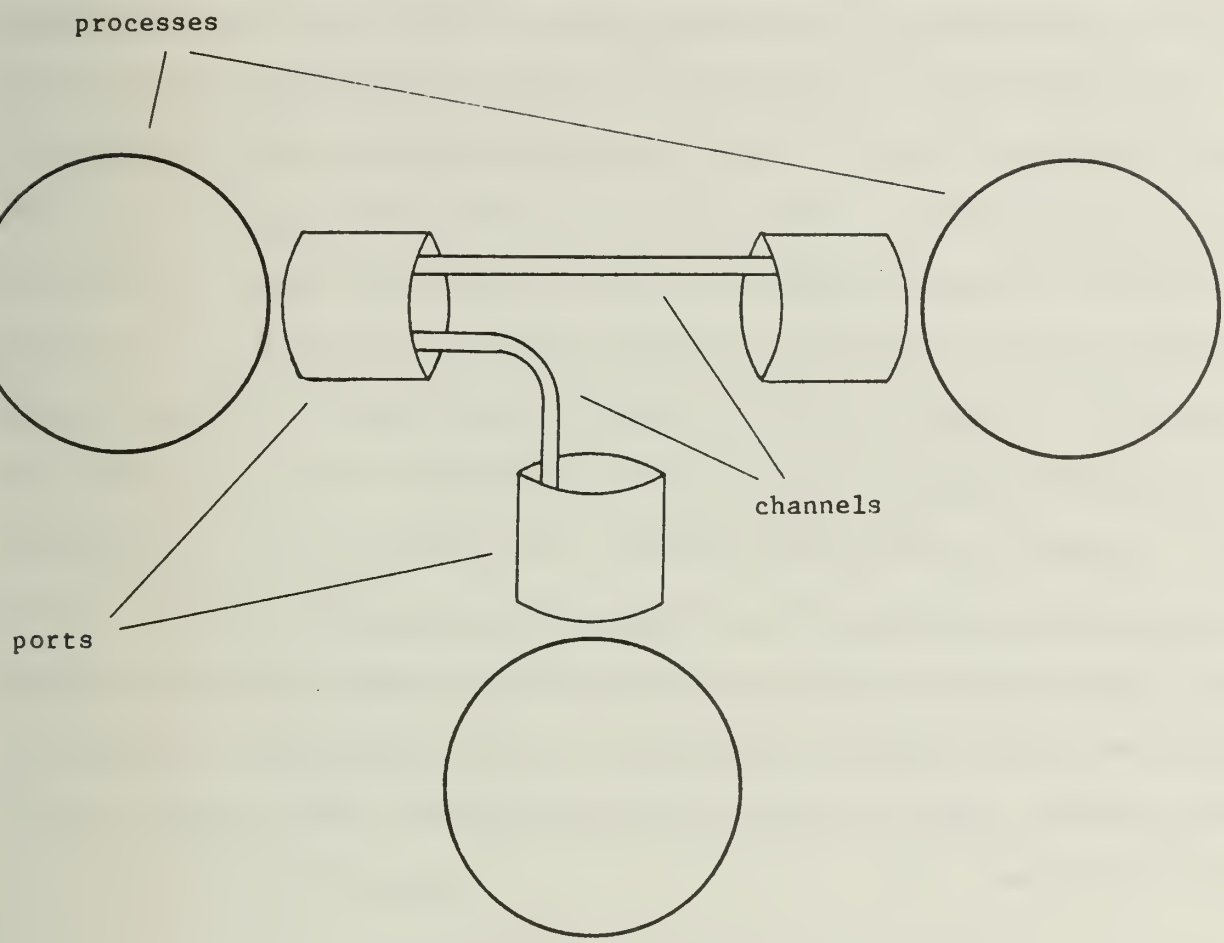

Figure 2: Process Communication.

The establishment of channels between ports is dynamic, changing as files are opened and closed, as interactive users log in and out, and as application programs make demands of the system. The efficient implementation of such a dynamic structure requires that ports have free access to one another. This, however, raises the possibility of access violations and error propagation. Such errors are confined to the system level by use of a 'ticket' mechanism which also facilitates governing of resource allocations.

5 Programming.

The implementation of Pathos in Path Pascal is based on the use of objects for the construction of synchronized data structures. Tickets and ports are the two main components of the message passing system. Tickets comprise a portable mechanism used to detect simple programming errors and to allocate resources both within this operating system and across the network to which it is connected. Tickets conceptually are protected process numbers. In the message passing system they are passed as parameters to system objects to check consistency of access within the operating system design. The process number also establishes a mapping between ports and processes. The communication controller manages the communication system by maintaining an array of pointers to ports; a process uses the port in the port array corresponding to its process number. The set of ports is a concurrently accessed data structure which defines the message transport mechanism. A port constitutes a recursively defined dynamic data structure which interfaces with other ports and with its associated process.

5.1 Tickets.

The ticket is implemented as an entry record in the 'conductor' object. The type definition of the ticket is exported from the conductor object, enabling each process to declare its own ticket and pass it to the conductor to be initialized and read. The contents of tickets cannot be written or directly read outside the conductor, ensuring that they can neither be modified nor forged. The conductor's basic operations on a ticket are 'stamp' for initialization and 'look' for reading. Because tickets map processes to ports and the communication controller establishes channel connections between ports, it is

necessary to control the order of ticket stamping at system initialization. A dummy operation, 'wait_stamp', is used in the conductor so that tickets are stamped in the correct sequence. This operation is executed by the communication controller immediately after it instantiates each process and allows some low level processes to make assumptions about port numbers for increased efficiency. When the instantiated process executes a stamp operation in its initialization sequence, 'wait_stamp' can proceed and release the communication controller to instantiate further processes as required. It has been suggested that 'wait_stamp' could have been avoided by passing tickets as parameters to processes. This would satisfy the protection uses of tickets but would reduce the generality of the mechanism for network applications.

Because only one process is 'stamp'ing at a time and no data in the conductor is used for 'look'ing at tickets, there are no exclusion requirements on the conductor's operations. The only sequencing constraint, that 'wait_stamp' cannot proceed until a 'stamp' has been executed, is easily incorporated into the object's Path Expression:

PATH (stamp; wait_stamp), look END

The entire object is straightforward to code once the path is complete. 'Stamp' writes the next ticket value into the ticket passed in as a parameter. 'Look' returns as its function value the value of the ticket it receives in the ticket passed in as a parameter. 'Wait_stamp' has no code and is used only for synchronization.

5.2 Ports.

The port is a bidirectional message passing object. Incoming messages are queued within the port until they are requested as input by the process associated with the port. Outgoing messages are passed directly through the object to the destination port where they become incoming messages to the destination process. The operations in the port are divided into two distinct, independent sets corresponding to the two message directions.

The set of operations which implement the outgoing direction of the port contains the two operations 'put' and 'channel_set'. The function of 'put' is to pass the message to the destination port. The input channel to destination port mapping is set by the operation 'channel_set'. The operations share the port mapping data structure and are synchronized by the following portion of the port's Path Expression.

1: (put, channel_set)

The incoming side of the port has three operations: 'supply', 'get' and 'q_length'. The operations 'supply' and 'get' manage the queue of incoming messages by attaching and removing messages respectively. A message is viewed as a resource to be managed by the port because only those messages received can be read. This is governed by the sequencing mechanism in the port's Path Expression: a 'supply' must precede every 'get'. The Path Expression also expresses the exclusion requirements on the incoming queue's data: the three operations share pointers for the message queue so only one of them can execute at a time.

1: (supply, get, q_length), (supply; get)

The operation 'q_length' is a function which returns the number of

messages on the incoming message queue. This operation is used by interrupt processes which cannot block and by the user supervisor process which uses input queue information to make scheduling decisions for user processes. A final operation is needed to initialize the port so it can recognize the ticket of its owner process. This is done by the operation 'owner_set' during the system initialization sequence before any processes are instantiated. No synchronization is required.

The Path Expression fragments described above are combined to produce the entire expression for the port object:

```
PATH 1:(put, channel_set), owner_set,
    1:(supply, get, q_length), (supply; get) END
```

The port_type object can be implemented based on the discussion above. The port_type in Pathos is 100 lines of Path Pascal [McKendry, 80].

6 Implementation.

The design phase of the system level of Pathos, including the specification of the file system and device drivers, required 6 man-weeks. Coding of the message passing subsystem, basic communication controller, and teletype drivers (a total of 900 lines of Path Pascal) took two weeks. Following a successful compilation, several runs on a Cyber 175 in a simulated environment showed obvious initialization errors. These runs, and the consequent changes, were completed in one day. The system was then transferred to a 'real' environment on an LSI-11 for final testing. Within 6 hours all known bugs were detected and located in the Path Pascal program. These bugs were corrected in the next 2 or 3 days.

Of the 9 bugs found in the system, 5 were synchronization errors. Of these, 4 were in the teletype drivers where semaphores had been used in the interests of 'efficiency'. These bugs were eliminated once the offending code was rewritten as a character queue object. The fifth synchronization error was due to misunderstanding of the Path Expression for the port object.

The author of Pathos has had four years experience in the design and implementation of systems with similar aims [McKendry, 78]. These systems were coded in high level assembly languages (e.g., PL-11 [Russell, 76]) using binary semaphores for synchronization. Debugging was at the machine language level, taking three to four man weeks for systems comparable to Pathos. Although the number of lines of code was similar, the number of bugs was almost an order of magnitude greater in the previous systems and some particularly complex synchronization sequences were never fully debugged or understood. The processes containing these sequences were replaced when the original programmer left the organization. Experience in the implementation of Pathos in Path Pascal so far indicates that most, if not all, of the pitfalls of semaphore synchronization and low level debugging have been overcome. The use of objects for encapsulation and Path Expressions for synchronization has improved structure and clarity to the extent that few bugs survive past the coding phase. These bugs are easily found through simulation and execution on the target machine. The ability of the kernel to do simple logical IO during debugging allows the use of the standard Pascal output routines for trace statements and, in the case of Pathos, eliminated entirely the need to debug at any level below the source language.

7 Results.

The most notable feature of the Pathos implementation is the simplicity of the program constructs used. No Path Expression is more complex than those described in this paper. Sequencing and resource restriction, together with the ability to construct nested refinements of synchronization constraints, appear to be sufficient for all forseeable Pathos situations. They allow precise constraints on sequencing and exclusion to be specified without entering into over-exclusion situations, a failing of some synchronization mechanisms [Hoare, 74]. There has been no need to control scheduling of processes entering or leaving objects. Despite the protestations of other authors [Bloom, 79], the simple demand scheduling implemented by the kernel is adequate; any excess processor time at the system level is consumed by the user supervisor. Unlike monitor based systems [Hoare, 74] there was no need for conditional or process state dependent synchronization.

The use of synchronization procedures in objects has been a source of some contention in the literature [Bloom, 79]. In Pathos this has been necessary in the conductor, where 'wait_stamp' synchronizes the stamping of tickets. A less obvious example is found in the message allocator, where nested objects and a Path Expression configured as a counting semaphore are used to circumvent the nested monitor problem. This problem arises in many synchronization schemes [Lister, 77]. The solution implemented using nested Path Expressions is simple and efficient and avoids complex extensions to the synchronization mechanism solely for this purpose. Apparent inefficiencies in the solution can easily be eliminated by an optimizing compiler, since the synchronization schemes are static at compile time. These two cases are the only instances of executable code whose explicit purpose is synchronization. All other code is written

without regard for synchronization or scheduling decisions.

The availability of objects and the ability to recursively define them had a strong influence on the design of Pathos, with the result that all requests by processes for global resources are calls on entry procedures in objects. Other than objects there is no data shared between processes. This greatly clarifies the structure of the system and, together with the lack of conditional synchronization, will aid in a proof of its correctness. The improved clarity is a major contributor to the lack of bugs in the system and the speed of the debugging phase of implementation.

8 Conclusion.

The evaluation of Path Pascal by experience has illustrated its usefulness for operating system construction. No direct comparisons with other languages have been made but the experiment has pointed out many strengths of Path Pascal in its simplicity, modularity, and ease of use. The influence of the Path Pascal object in system design has been beneficial to program structure. The Path Expressions used were easily developed and coded.

Often the intentions of a designer in adding new programming language features are clear to the designer but when the features are put to a practical test it is discovered that those intentions are not seen by programmers who may choose to emphasize features other than those originally considered important. Furthermore, a practical test may point out problems which were not forseen in the fitting of the features into the base language. Programming of Pathos did indeed lead to the uncovering of several of these problem areas. It became apparent, for example, that the desirability of using mutually recursive objects and procedures necessitated the introduction of some mechanism for varying the

order of declarations, such as a 'forward' object declaration. Problems such as this are annoying to a general user and, while such a user is the ultimate judge of the language, it is more desirable for the original implementors to make these discoveries themselves.

The Pathos experiment has been valuable in the illumination of strengths of Path Pascal and as experience in the use of its experimental features. The objectives of Path Expression synchronization were well understood before the experiment, but their ease of use, the reliability of the code produced, and the structural benefits of objects were beyond expectations. Such an experiment is to be recommended in the evaluation of new languages or features.

9 Acknowledgements.

This work was funded in part by NASA project (NSG 1471) and NSF project (MCS 77-09128).

10 References.

[Bloom, 79] Bloom, Tony, "Evaluating Synchronization Mechamnisms," SIGOPS Proc. of 7th Symp. on OS Principles, pp. 24-32, Pacific Grove, CA, December 10-12, 1979.

[Campbell & Kolstad, 79] Campbell, R. H. and R. B. Kolstad, "Path Expressions in Pascal," Fourth International Conference on Software Engineering, Munich, September 17-19, 1979.

[Dijkstra, 68] Dijkstra, E. W., "The Structure of the 'THE'-Multiprogramming System," CACM, Vol. 11, No. 5, pp. 341-346, May, 1968.

[Hewitt & Atkinson, 79] Hewitt, C., and Atkinson, R., "Specification and Proof Techniques for Serializers", IEEE Trans. on Software Engineering, Vol. SE-5, No. 1, pp. 10-23, January 1979.

[Hoare, 74] Hoare, C. A. R., "Monitors: An Operating System Structuring Concept," CACM, Vol. 17, No. 10, pp. 549-557, October, 1974.

[Lister, 77] Lister, A., "The Problem of Nested Monitor Calls," ACM Operating System Review, Vol. 11, No. 3, July 1977.

[Lauer & Needham, 78] Lauer, Hugh and R. Needham, "On the Duality of Operating System Structures," Second International Symposium on Operating Systems, IRIA, October, 1978.

[McKendry, 78] McKendry, M. S., "The Use of Monitors in Microprocessor Software Development," Euromicro Journal, Vol. 4, No. 5, September, 1978.

[McKendry, 80] McKendry, M. S., "Pathos: An Experiment to Evaluate Path Pascal," M. S. Thesis, University of Illinois at Urbana-Champaign, April, 1980.

[Parnas, 72] Parnas, D. L. "A technique for software module specification with examples," Comm. ACM 15, pp. 330-336, 1972.

[Russell, 76] Russell, R. D., "Experience in the Design, Implementation, and Use of PL-11, A Programming Language for the PDP-11," SIGPLAN Notices, Vol. 11, No. 4, April, 1976.

[Wirth, 77] Wirth, N., "Toward a Discipline of Real-Time Programming," CACM, Vol. 20, No. 8, pp. 577-583, August, 1977.

[Zelkowitz, et al., 79] Zelkowitz, M. V., Shaw, A. C., and Gannon, J. D., Principles of Software Engineering, Prentiss-Hall, 1979.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-80-1016	2.	3. Recipient's Accession No.

4. Title and Subtitle

PATHOS: A Path Pascal Operating System

5. Report Date
June 1980

6.

7. Author(s)
M. McKendry, R. Campbell, R. Kolstad

8. Performing Organization Rept. No. R-80-1016

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois U-C
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.
NASA NSG 1471
NSF MCS 77-09128

12. Sponsoring Organization Name and Address

NASA Langley Research Center
Hampton, Virginia 23665

National Science Foundation
Washington, DC 20550

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

This paper examines the use of Path Pascal in the implementation of Pathos, an example operating system. Path Pascal is an experimental programming language which includes concurrent processes, synchronization, and shared data objects combined with an encapsulation mechanism. Pathos is based on the canonical message passing operating system and was implemented as a practical test of Path Pascal. This paper reviews the principles of concurrent programming in Path Pascal and examines the influence of the language on operating system structure, modularity, and ease of program implementation and debugging.

17. Key Words and Document Analysis. 17a. Descriptors

Path Pascal
concurrent processes
synchronization
shared data objects

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
unlimited	UNCLASSIFIED	23
	20. Security Class (This Page) UNCLASSIFIED	22. Price